Examples

These examples use a `pageviews` stream and a `users` table.

> **Tip**
>
> The Stream Processing Cookbook [https://www.confluent.io/product/ksql/stream-processing-cookbook] contains ksqlDB recipes that provide in-depth tutorials and recommended deployment scenarios.

Creating streams

Prerequisite:

- The corresponding Kafka topics, named `pageviews` and `users`, must exist in your Apache Kafka® cluster.

Create a stream with three columns on the Kafka topic that is named `pageviews`.

ksqlDB can't infer the topic's data format, so you must provide the format of the values that are stored in the topic. In this example, the values format is `DELIMITED`.

```
CREATE STREAM pageviews
  (viewtime BIGINT,
   userid VARCHAR,
   pageid VARCHAR)
  WITH (KAFKA_TOPIC='pageviews',
        VALUE_FORMAT='DELIMITED');
```

Associate Kafka message keys

The previous statement doesn't make any assumptions about the Kafka message key in the underlying Kafka topic. But if the value of the message key in Apache Kafka® is the same as one of the columns defined in the stream in ksqlDB, you can provide this information in the WITH clause. For example, if the Apache Kafka® message key has the same value as the `pageid` column, you can write the CREATE STREAM statement like this:

```
CREATE STREAM pageviews
  (viewtime BIGINT,
   userid VARCHAR,
   pageid VARCHAR)
 WITH (KAFKA_TOPIC='pageviews',
       VALUE_FORMAT='DELIMITED',
       KEY='pageid');
```

Associate Apache Kafka® message timestamps

If you want to use the value of one of the columns as the Apache Kafka® message timestamp, you can provide this information to ksqlDB by using the WITH clause. The message timestamp is used in window-based operations in ksqlDB (such as windowed aggregations) and to support event-time based processing in ksqlDB. For example, if you want to use the value of the `viewtime` column as the message timestamp, you can rewrite the above statement like this:

```
CREATE STREAM pageviews
  (viewtime BIGINT,
   userid VARCHAR,
   pageid VARCHAR)
  WITH (KAFKA_TOPIC='pageviews',
        VALUE_FORMAT='DELIMITED',
        KEY='pageid',
        TIMESTAMP='viewtime');
```

Create tables

Prerequisite:

- The corresponding Apache Kafka® topics must already exist in your Apache Kafka® cluster.

Create a table with several columns. In this example, the table has columns with primitive data types, a column of `array` type, and a column of `map` type:

```
CREATE TABLE users
  (registertime BIGINT,
   gender VARCHAR,
   regionid VARCHAR,
   userid VARCHAR,
   interests array<VARCHAR>,
   contactinfo map<VARCHAR, VARCHAR>)
  WITH (KAFKA_TOPIC='users',
        VALUE_FORMAT='JSON',
        KEY = 'userid');
```

Note that specifying KEY is required in table declaration, see Key Requirements [../../developer-guide/syntax-reference/#key-requirements].

Working with streams and tables

Now that you have the `pageviews` stream and `users` table, take a look at some example queries that you can write in ksqlDB. The focus is on two types of ksqlDB statements: CREATE STREAM AS SELECT (CSAS) and CREATE TABLE AS SELECT (CTAS). For these statements, ksqlDB persists the results of the query in a new stream or table, which is backed by a Apache Kafka® topic.

Transforming

For this example, imagine you want to create a new stream by transforming `pageviews` in the following way:

- The `viewtime` column value is used as the Apache Kafka® message timestamp in the new stream's underlying Apache Kafka® topic.
- The new stream's Apache Kafka® topic has 5 partitions.

- The data in the new stream is in JSON format.
- Add a new column that shows the message timestamp in human-readable string format.
- The `userid` column is the key for the new stream.

The following statement generates a new stream, `pageviews_transformed` with the above properties:

```
CREATE STREAM pageviews_transformed
  WITH (TIMESTAMP='viewtime',
        PARTITIONS=5,
        VALUE_FORMAT='JSON') AS
  SELECT viewtime,
         userid,
         pageid,
         TIMESTAMPTOSTRING(viewtime, 'yyyy-MM-dd HH:mm:ss.SSS') AS
timestring
  FROM pageviews
  PARTITION BY userid
  EMIT CHANGES;
```

Use a `[ WHERE condition ]` clause to select a subset of data. If you want to route streams with different criteria to different streams backed by different underlying topics, e.g. content-based routing, write multiple ksqlDB statements as follows:

```
CREATE STREAM pageviews_transformed_priority_1
  WITH (TIMESTAMP='viewtime',
        PARTITIONS=5,
        VALUE_FORMAT='JSON') AS
  SELECT viewtime,
         userid,
         pageid,
         TIMESTAMPTOSTRING(viewtime, 'yyyy-MM-dd HH:mm:ss.SSS') AS
timestring
  FROM pageviews
  WHERE userid='User_1' OR userid='User_2'
  PARTITION BY userid
  EMIT CHANGES;
```

```
CREATE STREAM pageviews_transformed_priority_2
      WITH (TIMESTAMP='viewtime',
            PARTITIONS=5,
            VALUE_FORMAT='JSON') AS
  SELECT viewtime,
         userid,
         pageid,
         TIMESTAMPTOSTRING(viewtime, 'yyyy-MM-dd HH:mm:ss.SSS') AS
timestring
  FROM pageviews
  WHERE userid<>'User_1' AND userid<>'User_2'
  PARTITION BY userid
  EMIT CHANGES;
```

Joining

When joining objects the number of partitions in each must be the same. You can use ksqlDB itself to create re-partitioned streams/tables as required. In this example, you join `users` to the `pageviews_transformed` topic, which has 5 partitions.

First, generate a `users` topic with a partition count to match that of `pageviews_transformed`:

```
CREATE TABLE users_5part
    WITH (PARTITIONS=5) AS
    SELECT * FROM USERS
    EMIT CHANGES;
```

Now you can use the following query to create a new stream by joining the `pageviews_transformed` stream with the `users_5part` table.

```
CREATE STREAM pageviews_enriched AS
  SELECT pv.viewtime,
         pv.userid AS userid,
         pv.pageid,
         pv.timestring,
         u.gender,
         u.regionid,
```

```
        u.interests,
        u.contactinfo
  FROM pageviews_transformed pv
  LEFT JOIN users_5part u ON pv.userid = u.userid
  EMIT CHANGES;
```

Note that by default all of the Apache Kafka® topics are read from the current offset (the latest available data). But in a stream-table join, the table topic is read from the beginning.

Aggregating, windowing, and sessionization

Watch the screencast on aggregation [https://www.youtube.com/embed/db5SsmNvej4].

Now assume that you want to count the number of pageviews per region. Here is the query that would perform this count:

```
CREATE TABLE pageviews_per_region AS
  SELECT regionid,
         count(*)
  FROM pageviews_enriched
  GROUP BY regionid
  EMIT CHANGES;
```

The above query counts the pageviews from the time you start the query until you terminate the query. Note that we used CREATE TABLE AS SELECT statement here since the result of the query is a ksqlDB table. The results of aggregate queries in ksqlDB are always a table, because ksqlDB computes the aggregate for each key (and possibly for each window per key) and updates these results as it processes new input data.

ksqlDB supports aggregation over WINDOW too. Here's the previous query changed to compute the pageview count per region every 1 minute:

```
CREATE TABLE pageviews_per_region_per_minute AS
  SELECT regionid,
         count(*)
```

```
  FROM pageviews_enriched
  WINDOW TUMBLING (SIZE 1 MINUTE)
  GROUP BY regionid
  EMIT CHANGES;
```

If you want to count the pageviews for only "Region_6" by female users for every 30 seconds, you can change the above query as the following:

```
CREATE TABLE pageviews_per_region_per_30secs AS
  SELECT regionid,
         count(*)
  FROM pageviews_enriched
  WINDOW TUMBLING (SIZE 30 SECONDS)
  WHERE UCASE(gender)='FEMALE' AND LCASE(regionid)='region_6'
  GROUP BY regionid
  EMIT CHANGES;
```

UCASE and LCASE functions in ksqlDB are used to convert the values of gender and regionid columns to upper and lower case, so that you can match them correctly. ksqlDB also supports LIKE operator for prefix, suffix and substring matching.

ksqlDB supports HOPPING windows and SESSION windows too. The following query is the same query as above that computes the count for hopping window of 30 seconds that advances by 10 seconds:

```
CREATE TABLE pageviews_per_region_per_30secs10secs AS
  SELECT regionid,
         count(*)
  FROM pageviews_enriched
  WINDOW HOPPING (SIZE 30 SECONDS, ADVANCE BY 10 SECONDS)
  WHERE UCASE(gender)='FEMALE' AND LCASE (regionid) LIKE '%_6'
  GROUP BY regionid
  EMIT CHANGES;
```

The next statement counts the number of pageviews per region for session windows with a session inactivity gap of 60 seconds. In other words, you are *sessionizing* the input data and then perform the counting/aggregation step per region.

```
CREATE TABLE pageviews_per_region_per_session AS
  SELECT regionid,
         count(*)
  FROM pageviews_enriched
  WINDOW SESSION (60 SECONDS)
  GROUP BY regionid
  EMIT CHANGES;
```

Sometimes, you may want to include the bounds of the current window in the result so that it is more easily accessible to consumers of the data. The following statement extracts the start and end time of the current session window into fields within output rows.

```
CREATE TABLE pageviews_per_region_per_session AS
  SELECT regionid,
         windowStart(),
         windowEnd(),
         count(*)
  FROM pageviews_enriched
  WINDOW SESSION (60 SECONDS)
  GROUP BY regionid
  EMIT CHANGES;
```

Working with arrays and maps

The `interests` column in the `users` table is an `array` of strings that represents the interest of each user. The `contactinfo` column is a string-to-string `map` that represents the following contact information for each user: phone, city, state, and zipcode.

> **Tip**
>
> If you are using `ksql-datagen`, you can use `quickstart=users_` to generate data that include the `interests` and `contactinfo` columns.

The following query creates a new stream from `pageviews_enriched` that includes the first interest of each user along with the city and zipcode for each user:

```
CREATE STREAM pageviews_interest_contact AS
  SELECT interests[0] AS first_interest,
         contactinfo['zipcode'] AS zipcode,
         contactinfo['city'] AS city,
         viewtime,
         userid,
         pageid,
         timestring,
         gender,
         regionid
  FROM pageviews_enriched
  EMIT CHANGES;
```

Run ksqlDB Statements From the Command Line

In addition to using the ksqlDB CLI or launching ksqlDB servers with the `--queries-file` configuration, you can also execute ksqlDB statements directly from your terminal. This can be useful for scripting. The following examples show common usage.

Use pipelines (`|`) to run ksqlDB CLI commands:

```
echo -e "SHOW TOPICS;\nexit" | ksql
```

Use the Bash here document [http://tldp.org/LDP/abs/html/here-docs.html] (`<<`) to run ksqlDB CLI commands:

```
ksql <<EOF
SHOW TOPICS;
SHOW STREAMS;
exit
EOF
```

Uses a Bash here string [http://tldp.org/LDP/abs/html/x17837.html] (`<<<`) to run ksqlDB CLI commands on an explicitly defined ksqlDB Server endpoint:

```
ksql http://localhost:8088 <<< "SHOW TOPICS;
SHOW STREAMS;
```

```
exit"
```

Use the `RUN SCRIPT` command to create a stream from a predefined `application.sql` script, and run a query by using the Bash here document [http://tldp.org/LDP/abs/html/here-docs.html] (`<<`) feature:

```
cat /path/to/local/application.sql
CREATE STREAM pageviews_copy AS SELECT * FROM pageviews EMIT CHANGES;
```

```
ksql http://localhost:8088 <<EOF
RUN SCRIPT '/path/to/local/application.sql';
exit
EOF
```

Note

The `RUN SCRIPT` command supports a subset of ksqlDB CLI commands, including running DDL statements (CREATE STREAM, CREATE TABLE), persistent queries (CREATE STREAM AS SELECT, CREATE TABLE AS SELECT), and setting configuration options (SET statement). Other statements and commands such as `SHOW TOPICS` and `SHOW STREAMS` are ignored.

Page last revised on: 2019-12-17

Last update: 2019-12-17